Solving Large-Scale Minimum-Weight Triangulation Instances to Provable Optimality

Andreas Haas
Department of Computer Science, TU Braunschweig
Braunschweig, Germany
haas@ibr.cs.tu-bs.de

Abstract

We consider practical methods for the problem of finding a minimum-weight triangulation (MWT) of a planar point set, a classic problem of computational geometry with many applications. While Mulzer and Rote proved in 2006 that computing an MWT is NP-hard, Beirouti and Snoeyink showed in 1998 that computing provably optimal solutions for MWT instances of up to 80,000 *uniformly distributed* points is possible, making use of clever heuristics that are based on geometric insights. We show that these techniques can be refined and extended to instances of much bigger size and different type, based on an array of modifications and parallelizations in combination with more efficient geometric encodings and data structures. As a result, we are able to solve MWT instances with up to 30,000,000 uniformly distributed points in less than 4 minutes to provable optimality. Moreover, we can compute optimal solutions for a vast array of other benchmark instances that are *not* uniformly distributed, including normally distributed instances (up to 30,000,000 points), all point sets in the TSPLIB (up to 85,900 points), and VLSI instances with up to 744,710 points. This demonstrates that from a practical point of view, MWT instances can be handled quite well, despite their theoretical difficulty.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases computational geometry, minimum-weight triangulation

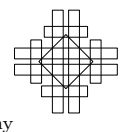
Digital Object Identifier 10.4230/LIPIcs.SoCG.2018.44

Related Version A full version of this paper is available at `https://arxiv.org/abs/1802.06415`

Acknowledgements I want to thank Sándor Fekete and Victor Alvarez for useful discussions and suggestions that helped to improve the presentation of this paper.

1 Introduction

Triangulating a set of points in the plane is a classic problem in computational geometry: given a planar point set S, find a maximal set of non-crossing line segments connecting the points in S. Triangulations have many real-world applications, for example in terrain modeling, finite element mesh generation and visualization. In general, a point set has exponentially many possible triangulations and a natural question is to ask for a triangulation that is optimal with respect to some optimality criterion. Well known and commonly used is the Delaunay triangulation, which optimizes several criteria at the same time: it maximizes the minimum angle and minimizes both the maximum circumcircle and the maximum smallest enclosing circle of all triangles. Another natural optimality criterion, and the one we are considering in this paper is minimizing the total weight of the resulting triangulation, i.e., minimizing the sum of the edge lengths.



The minimum-weight triangulation (MWT) is listed as one of the open problems in the famous book from 1979 by Garey and Johnson on NP-completeness [10]. The complexity status remained open for 27 years until Mulzer and Rote [19] finally resolved the question and showed NP-hardness of the MWT problem.

Independently, Gilbert [11] and Klincsek [17] showed that, when restricting it to simple polygons, the MWT problem can be solved in $O(n^3)$-time with dynamic programming. The dynamic programming approach can be generalized to polygons with k inner points. Hoffmann and Okamoto [16] showed how to obtain the MWT of such a point set in $O(6^k n^5 \log n)$-time. Their algorithm is based on a polygon decomposition through x-monotone paths. Grantson et al. [12] improved the algorithm to $O(n^4 4^k k)$ and showed another decomposition strategy based on cutting out triangles [13] which runs in $O(n^3 k! k)$-time.

A promising approach are polynomial-time heuristics that either include or exclude edges with certain properties from any minimum weight triangulation. Das and Joseph [6] showed that every edge in a minimum weight triangulation has the *diamond property*. An edge e cannot be in MWT(S) if both of the two isosceles triangles with base e and base angle $\pi/8$ contain other points of S. Drysdale et al. [9] improved the angle to $\pi/4.6$. This property can exclude large portions of the edge set and works exceedingly well on uniformly distributed point sets, for which only an expected number of $O(n)$ edges remain. Dickerson et al. [8, 7] proposed the *LMT-skeleton heuristic*, which is based on a simple local-minimality criterion fulfilled by every edge in MWT(S). The LMT-skeleton algorithm often yields a connected graph, such that the remaining polygonal faces can be triangulated with dynamic programming to obtain the minimum weight triangulation.

Especially the combination of the diamond property and the LMT-skeleton made it possible to compute the MWT for large, well-behaved point sets. Beirouti and Snoeyink [3, 2] showed an efficient implementation of these two heuristics and they reported that their implementation could compute the exact MWT of 40,000 uniformly distributed points in less than 5 minutes and even up to 80,000 points with the improved diamond property.

Our contributions:

- We revisit the diamond test and LMT-skeleton based on Beirouti's and Snoeyink's [3, 2] ideas and describe several improvements. Our bucketing scheme for the diamond test does not rely on a uniform point distribution and filters more edges. For the LMT-skeleton heuristic we provide a number of algorithm engineering modifications. They contain a data partitioning scheme for a parallelized implementation and several other changes for efficiency. We also incorporated an improvement to the LMT-skeleton suggested by Aichholzer et al. [1].
- We implemented, streamlined and evaluated our implementation on various point sets. For the uniform case, we computed the MWT of 30,000,000 points in less than 4 minutes on commodity hardware; the limiting factor arose from the memory of a standard machine, not from the runtime. We achieved the same performance for normally distributed point sets. The third class of point sets were benchmark instances from the TSPLIB [20] (based on a wide range of real-world and clustered instances) and the VLSI library. These reached a size up to 744,710 points. This shows that from a practical point of view, wide range of huge MWT instances can be solved to provable optimality with the right combination of theoretical insight and algorithm engineering.

2 Preliminaries

Let S be a set of points in the euclidean plane. A triangulation T of S is a maximal planar straight-line graph with vertex set S. The weight $w(e)$ of an edge e is its euclidean length. A minimum-weight triangulation $\mathrm{MWT}(S)$ minimizes the total edge weight, i.e., $\sum_{e \in E(T)} w(e)$.

An edge e is *locally minimal* with respect to some triangulation $T(S)$ if either

(i) e is an edge on the convex hull of S, or

(ii) the two triangles bordering e in $T(S)$ form a quadrilateral q such that q is either not convex or e is the shorter diagonal of the two diagonals e and e' in q, i.e., $w(e) \leq w(e')$.

A triangulation T is said to be a *locally minimal triangulation* if every edge of T is locally minimal, i.e., the weight of T cannot be improved by edge flipping. A pair of triangles witnessing local minimality for some edge e in some triangulation is called a *certificate* for e. An *empty triangle* is a triangle that contains no other points of S except for its three vertices.

3 Previous tools

3.1 Diamond property

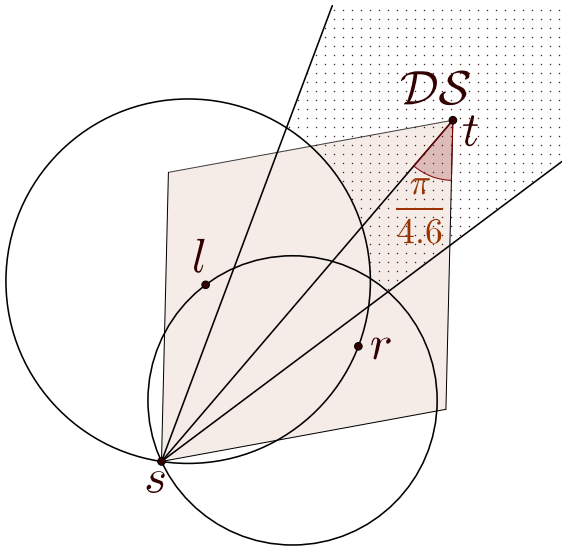


Figure 1 Points l and r induce a region $\mathcal{DS}$ such that all edges $e = st$ with $t \in \mathcal{DS}$ fail the diamond test. $\mathcal{DS}$ is called a dead sector (dotted area).

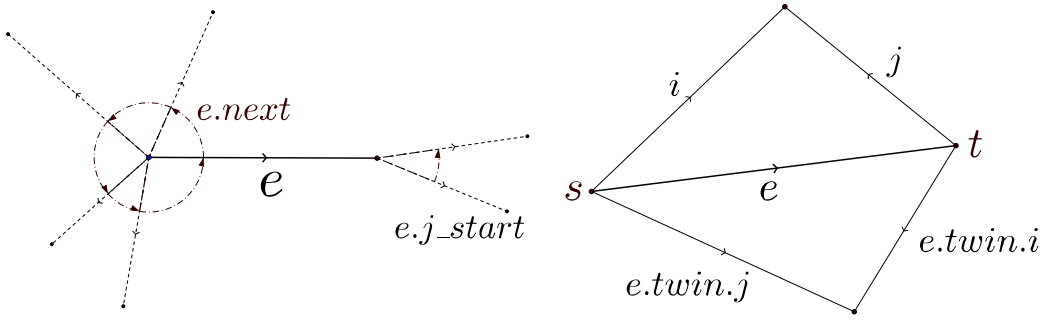
The diamond property can be used as a first step to exclude a large part of the edge set from any further consideration. A brute-force solution to test the diamond property for each edge takes $\Theta(n^3)$ time and is inefficient. To accelerate the test, Beirouti and Snoeyink [3] use a bucketing scheme based on a uniform grid with the grid size chosen such that on expectation a constant number of points lie in each cell. In order to quickly discard whole cells, they make use of *dead sectors*, which are illustrated in Figure 1. Suppose we want to test all edges with source s. If points l, r are already processed and known then all edges st with $t \in \mathcal{DS}$ will fail the diamond test, because l and r lie in the left, resp. right isosceles triangle. The boundary of a single dead sector depends on the angle and length of sl and sr; for multiple sectors it can be quite complicated. For each point s, all edges st that pass the

test are computed by first considering all other points in the cell containing s. Intersections of dead sectors with neighboring cells are computed and based on the result further cells are considered until all cells can be excluded by dead sectors.

3.2 LMT-skeleton

The LMT-skeleton was proposed by Dickerson et al. [8, 7], it is a subset of the minimum weight triangulation. The key observation is that MWT(S) is a locally minimal triangulation, i.e., no edge in MWT(S) can be flipped to reduce the total weight.

The LMT-skeleton algorithm eliminates all edges that have no certificate, i.e., for each edge e all pairs of empty triangles bordering e are examined until a certificate is found or no pairs are left. Eliminating e can invalidate previously found certificates. The remaining edges that are not intersected by any other remaining edge form the LMT-skeleton; they must be in all locally minimal triangulations.



(a) Half-edge e.

(b) i and j point to the current certificate edges.

Figure 2 Representation of half-edge e.

In order to avoid the $O(n^3)$ space required to store all empty triangles Beirouti and Snoeyink [3] propose a data structure based on half-edges. Half-edges store several pointers: a pointer to the target vertex and the twin half-edge; a pointer to the next edge in counterclockwise order around the source; and three additional pointers to scan for empty triangles (i, j, j_start); see Figure 2 for an illustration. A status flag indicates whether an edge is *possible, impossible* or *certain*. Furthermore, three additional pointers (*rightPoly, leftPoly, polyWeight*) are stored and used for the subsequent polygon triangulation step.

At the heart of the LMT-skeleton heuristic lies the `Advance` function, see Algorithm 1. `Advance` basically rotates edge i and j in counterclockwise order such that they form an

Algorithm 1: Advance. Adapted from [2] (Changed notation and corrected an error).

```
Function Advance(e)
    repeat
        while e.i.target is not left of e.j do
            e.i ← e.i.next ;
        while e.i.target is left of e.j do
            e.j ← e.j.next ;
    until e.i.target = e.j.target;
```

empty triangle if they point to the same vertex, i.e., $i.target = j.target$. Pointers i and j are initialized to $e.next$ resp. $e.j\_start$. The algorithm to find certificates is built on top of `Advance`. All pairs of triangles can be traversed by repeatedly calling `Advance` on half-edge e and e's twin in fashion similar to a nested loop. The "loop" is stopped when a certificate is found and can be resumed when the certificate becomes invalid. [2, 3]

After initializing the half-edge data structure, their implementation pushes all edges on a stack (sorted with longest edge on top) and then processes edges in that order. If for an edge e no certificate is found, an intersection test determines if e lies on the convex hull or if e is impossible. If e is detected to be impossible, a local scan restacks all edges with e in their certificate. After the stack is empty, all edges that remain possible and that have no crossing edges are marked as *certain*.

4 Our improvements and optimizations

4.1 Diamond property

For a uniformly distributed point set S with $|S| = n$ points, the expected number of edges to pass the diamond test is only $O(n)$. More precisely, Beirouti and Snoeyink [3] state that the number is less than $3\pi n/\sin(\alpha)$, where α is the base angle for the diamond property. We were able to tighten this value.

▶ **Theorem 1.** *Let S be a uniformly distributed point set in the plane with $|S| = n$ and let $\alpha \leq \pi/3$ be the base angle for the diamond property. Then the expected number of edges that pass the diamond test is less than $3\pi n/\tan(\alpha)$.*

Proof. Fix an arbitrary point $s \in S$ and consider the remaining points t_i , $0 \leq i \leq n-2$ in order of increasing distance to s. Edge $e_i := st_i$ fulfills the diamond property if at least one of the two corresponding isosceles triangles is empty, i.e., it contains non of the i points $t_0, \ldots, t_{i-1}$.

For any given distance r, each triangle has area $A = 1/4\tan(\alpha)r^2$. The points are uniformly distributed in the circle centered at s with radius r, thus the probability p that a fixed point lies in a fixed triangle is $p = A/\pi r^2 = \tan(\alpha)/4\pi$. Each triangle is empty with probability $(1-p)^i$. The whole diamond is empty with probability $(1-2p)^i$. It follows that at least one of the triangles is empty with probability $2(1-p)^i - (1-2p)^i$.

Let X_i be the indicator variable of the event that edge e_i fulfills the diamond property. Then $X = \sum_0^{n-2} X_i$ is the number of outgoing edges that pass the diamond test. By linearity of expectation and the geometric series, the expected value of X is bounded by

$$E[X] = \sum_{i=0}^{n-2} E[X_i] < \sum_{i=0}^{\infty} 2(1-p)^i - (1-2p)^i = \frac{2}{p} - \frac{1}{2p} = \frac{3}{2p} = \frac{6\pi}{\tan(\alpha)}$$

If we apply the same argument to each point in S, we are counting each edge twice. Hence the number of edges that pass the diamond test with base angle a is less than $3\pi n/\tan(\alpha)$. ◀

For $\alpha = \pi/4.6$ we get a value less than 11.5847, which is very close to the values observed and achieved by our implementation; see Table 1 in Section 5. In contrast, the value achieved by the implementation of Beirouti and Snoeyink is ≈ 14.3 [3].

4.2 Dead sectors and bucketing

Our bucketing scheme is based on the same idea of dead sectors as described by Snoeyink and Beirouti [3]. Our implementation differs in two points. Despite being simpler, it has

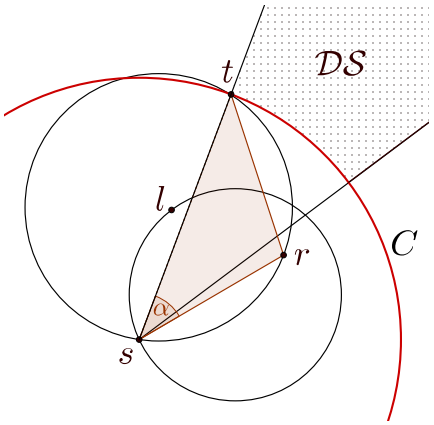


Figure 3 Simplified dead sector $\mathcal{DS}$ is bounded by two rays and circle C. C is induced by the longer of the two edges sl resp. sr and angle α.

higher accuracy and it can easily be integrated into common spatial data structures; such as quadtrees, kd-trees and R-trees. Therefore, it is not limited to uniformly distributed point sets.

In order to avoid storing complicated sector boundaries, we simplify the shape. Instead of bounding a sector $\mathcal{DS}$ by two circles as illustrated in Figure 1, we only use a single big circle C with center s at the expense of losing a small part of $\mathcal{DS}$. This allows a compact representation of dead sectors as a triple of three numbers: an interval consisting of two polar angles and a squared radius; see Figure 3.

The main ingredient for our bucketing scheme is a spatial search tree with support for incremental nearest neighbor searches, such as quadtrees, kd-trees or R-trees. A spatial search tree hierarchically subdivides the point set into progressively finer bounding boxes/rectangles until a predefined threshold is met. Incremental nearest neighbor search queries allow to traverse all nearest neighbors of a point in order of increasing distance. Such queries can easily be implemented by utilizing a priority queue that stores all tree nodes encountered during tree traversal together with the distances to their resp. bounding box (see Hjaltason and Samet [15]).

Pruning tree nodes whose bounding box lie in dead sectors is rather simple as follows: consider a nearest neighbor query for point s: when we are about to push a new node n into the priority queue, we compute the smallest polar angle interval I that encloses the bounding box of n and discard n if I is contained in the dead sectors computed so far. The interval of a bounding box is induced by the two extreme corners as seen from s, i.e., the leftmost and the rightmost corner.

Because nearest neighbors and tree nodes are processed in order of increasing distance, we can store sectors in two stages. On creation, they are inserted into a FIFO-queue; later only the interval component is inserted in a search filter used by the tree. The queue can be seen as a set of pending dead sectors with an attached activation distance δ. As soon as we process a point t with $\mathrm{d}(s,t) > \delta$ we can insert the corresponding interval into our filter.

This reduces the data structure used for the filter to a simple set of sorted non-overlapping intervals consisting of polar angles. Overlapping intervals are merged on insertion, which reduces the maximal number of intervals that need to be tracked at the same time to a very small constant[1].

[1] The exact value is 15 in our case, but it depends on an additional parameter and implementation details.

This leaves the issue of deciding which points are used to construct dead sectors. We store all points encountered during an incremental search query in an ordered set N sorted by their polar angle with respect to s. Every time we find a new point t, it is inserted into N and dead sectors are computed with the predecessor and the successor of t in N. There is no need to construct sectors with more than the direct predecessor and successor, because sectors between all adjacent pairs of points in N were already constructed on earlier insertions. Computing the activation distance for new sectors only requires a single multiplication of the current squared distance to t with a precomputed constant. Additionally, the diamond property of edge st is tested against a subset of N.

If we apply the above procedure to every single point, we generate each edge twice, once on each of the two endpoints. Therefore, we output only those edges $e = st$ such that $s < t$, i.e., s is lexicographically smaller than t. As a consequence, we can exclude a part of the left half-space right from the beginning by inserting an initial dead sector $\mathcal{DS}_0 = (1/2\pi + \alpha, 3/2\pi - \alpha)$ at distance 0. Points in the two wedges $(1/2\pi, 1/2\pi + \alpha]$ and $[3/2\pi - \alpha, 3/2\pi]$ are specially treated because they are still useful to generate dead sectors for the right half-space.

In order to increase cache efficiency we store the point set in a spatially sorted array. The points are ordered along a Hilbert curve, but the choice of a particular space-filling curve is rather arbitrary. Our spatial tree implementation is a quadtree that is built on top of that array during the sorting step. Profiling suggests the memory layout of the tree nodes is not important. We apply the diamond test to every single point and we can freely choose the order in which we process them. The points are spatially sorted and processed in this order, which leads to similar consecutive search paths in the tree and therefore most nodes are already in the CPU cache.

In order to avoid the expensive transcendental `atan2` function for polar angle computations, we can use any function that is monotonic in the polar angle for comparisons between different angles. One such function, termed *pseudo-angle*, was described by Moret and Shapiro [18]. The basic idea is to measure arc lengths on the L_1 unit circle, instead of the L_2 unit circle. With some additional transformations, the function can be rewritten to $\text{sign}(y)(1 - x/(|x| + |y|))$, where we define $\text{sign}(0) =: 1$. This function has the same general structure as `atan2`: a monotonic increase in the intervals $[0, \pi]$, $(\pi, 2\pi)$ and a discontinuity at π, with a jump from positive to negative values. Additionally, it gives rise to a one-line implementation (see the full version), which gets compiled to branch-free code.

4.3 LMT-skeleton

For "nicely" distributed point sets, a limiting factor of the heuristic is the space required to store the half-edge data structure in memory. In order to save some space we removed three variables from the original description (*rightPoly, leftPoly, polyWeight*). They serve no purpose until after the heuristic, when they are used for the polygon triangulation step (therefore, reducing cache-efficiency and wasting space). For edges marked *impossible* (typically the majority), they are never used at all; for the remaining edges they can be stored separately as soon as needed. We further reduce storage overhead by storing all edges in a single array sorted by source vertex (also known as a *compressed sparse row graph*). As a consequence, spatial ordering of the vertices carries over to the edge array. All outgoing edges of a single vertex are still radially sorted. In addition to the statuses *possible*, *certain*, *impossible*, we store whether an edge lies on the convex hull.

As mentioned in Section 3, certificates are found by utilizing `Advance` in fashion of a nested loop. It is crucial to define one half-edge of each pair as the primary one to distinguish

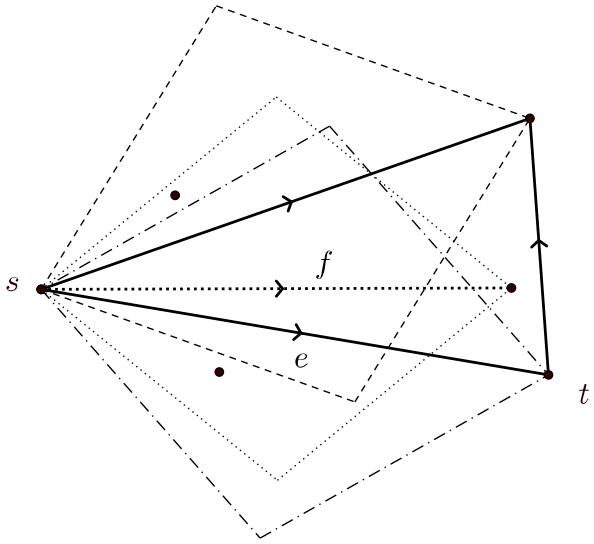


Figure 4 Edge f does not have the diamond property and in turn the `Advance` function fails: it stops with a non-empty triangle.

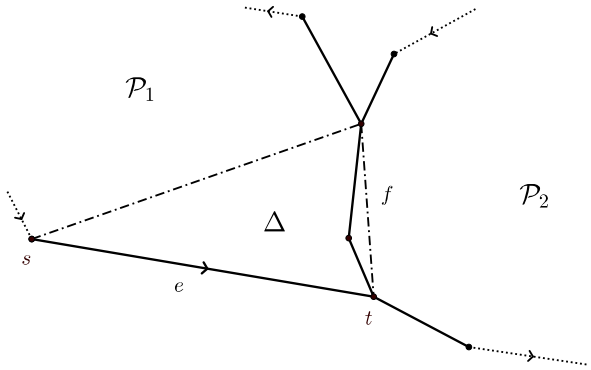


Figure 5 `Advance` can return non-empty triangle Δ which is part of two adjacent polygonal faces.

which half-edge corresponds to the outer resp. inner "loop". The choice is arbitrary as long as it is consistent throughout the execution of the algorithm.

Another problem that went unnoticed emerges when the diamond test and LMT-skeleton are combined. In this case `Advance` does not guarantee to find empty triangles; it may stop with non-empty triangles due to missing incident edges. An example is shown in Figure 4, where all edges with the exception of f pass the diamond test; calling `Advance` on e yields a non-empty triangle.

Fortunately, the side effect of wrong certificates is rather harmless. In the worst-case an otherwise impossible edge stays possible, which in turn may prevent other edges from being marked *certain*, however, no edge will incorrectly be marked *certain*. Even though wrong certificates occur frequently, we observed them to be of transient nature because some certificate edge itself becomes impossible later in the heuristic. Therefore, we still use `Advance` in our implementation to find certificates. However, it is important to keep in mind that the function can fail. Beirouti [2] states that they also use `Advance` to scan for empty triangles in simple polygons during the dynamic programming step after the LMT-skeleton. Even then `Advance` can fail by returning triangles that are part of two adjacent simple

Algorithm 2: Refactored LMT-skeleton algorithm.

```
begin
    Init data structures ;
    ST ← EmptyEdges(S)\ CH (S) ;                          /* Stack ST */
    LMT-Loop (ST) ;
    foreach Possible primary half-edge e do
        if ¬HasIntersections (e) then
            Mark e as certain;
Function LMT-Loop(ST)
    while ST is not empty do
        e ← Pop (ST) ;
        if e has no certificate then
            RestackEdges (e) ;    /* Push edges with e in their certificate.
             */
            Mark e as impossible;
```

polygonal faces; see Figure 5 for an example. In contrast to wrong certificates, this will lead to catastrophic results and cannot be ignored.

Pseudocode for our implementation is given in Algorithm 2. In essence it is still the same as given by Beirouti and Snoeyink [3], however, with some optimizations applied. First, the convex hull edges are implicitly given during initialization of the *j_start*-pointers and can be marked as such without any additional cost. Determining the convex hull edges beforehand allows to remove the case distinction inside the `LMT-Loop`, i.e., it removes all intersection tests that are applied to impossible edges. Secondly, sorting the stack by edge length destroys spatial ordering and the loss of locality of reference outweighs all gains on modern hardware. Without sorting, it is actually not necessary to push all edges onto the stack upfront. Lastly, with proper partitioning of the edges, the `LMT-Loop` can be executed in parallel – described in more detail in Section 4.4.

Additionally, we incorporated an improvement to the LMT-skeleton suggested by Aichholzer et al. [1]. Consider a certificate for an edge e, i.e., a quadrilateral q_e such that e is locally minimal w.r.t. q_e. It is only required that the four certificate edges $f_i \in q_e$ are not *impossible*, that is, edge f_i is either on the convex hull or in turn has some certificate q_i. Notice that q_i and q_e may not share a common triangle. However, if for edge f_i there is no such certificate q_i that shares a triangle with q_e, then edge e cannot be in any locally minimal triangulation and e can be marked *impossible*.

The improved LMT-skeleton is computationally much more expensive. Consider the case in which edge $e = (s, t)$ becomes impossible. In order to find invalid certificates, it is no longer sufficient to scan only those edges incident to either s or t. In addition to edges of the form (s, u), resp. (t, u), we also have to check all edges incident to any adjacent vertex u for invalid certificates. Because edges do not store the certificates for their certificate it gets even worse: we cannot know if an edge has to be restacked and we must restack and recheck all of them. Another consequence is that we cannot resume the traversal of triangles for any edge f_i, because we do not know where we stopped the last time.

We are left with a classic space-time trade-off and we chose not to store any additional data. Instead we apply the improved LMT-heuristic only to edges surviving an initial round of the normal LMT-heuristic.

4.4 Parallelization

Because the LMT-heuristic performs only local changes, most of the edges can be processed in parallel without synchronization. Problems occur only if adjacent edges are processed concurrently (for the improved LMT-skeleton this is unfortunately not true, because marking an edge *impossible* affects a larger neighborhood of edges). In order to parallelize the normal LMT-heuristic, we implemented a solution based on data partitioning without any explicit locking.

We cut the vertices V into two disjoint sets $V = V_1 \cup V_2$ and process only those edges with both endpoints in V_1 (resp. V_2) in parallel. Define X as the cut set $\{\{s,t\} \in E \mid s \in V_1, t \in V_2\}$, i.e., all edges with one endpoint in V_1 and the other in V_2. While edges in $E(V_1)$ resp. $E(V_2)$ are processed in parallel by two threads, edges in X are accessed read-only by both threads and are handled after both threads join. This way we never process two edges with a common endpoint in parallel.

This leaves the question of how to partition the vertices into two disjoint sets. Recall that all vertices are stored in contiguous memory and are sorted in Hilbert order. A split in the middle of the array partitions the points into two sets that are separated by a rather simple curve. Therefore, the cut set is likely to be small. Our half-edge array is sorted by source vertex, i.e., getting all edges with a specific source vertex in either half of the partition is trivial. Deciding if an edge $e = (s,t)$ is in the cut set consists of two comparisons of pointer t against the lower and upper bound of the vertex subset. Furthermore, with the fair assumption that the average degree of vertices is the same in both partitions, we obtain perfectly balanced partitions w.r.t. the number of edges.

In order to avoid a serial scan at the top, we push the actual work of computing X down to the leaves in the recursion tree. Scanning of the half-edge array starts at the leave nodes: processing of half-edges that belong to some cut set is postponed, instead they are passed back to the parent node. The parent in turn scans the edges it got from its two children, processes all edges it can and passes up the remaining ones. In other words, the final cut set X bubbles up in the tree, while all intermediate cuts are never explicitly computed. The edges passed up from a node typically contain half-edges of several higher-level cuts. This way, partitioning on each level of the recursion tree only takes constant time, while the actual work is fully parallelized at the leaf level.

Experiments and observations indicate that on large, uniformly distributed point sets approximately 0.15% of all edges make it back to the root node, i.e., the amount of serial processing is low and the approach scales well. On degenerate instances it can perform poorly; e.g. if all points lie on a circle, then half of the edges will be returned to the root. For such cases, the code could be extended to repartition the remaining edges with another set of cuts.

After the LMT-heuristic completes, we are left with many polygonal faces that still need to be triangulated. Our implementation traverses the graph formed by the edges with one producer thread in order to collect all faces and multiple consumer threads to triangulate them with dynamic programming.

5 Computational results

Computations were performed on a machine with an Intel i7-6700K quad-core and 64GB memory. The code was written in C++ and compiled with gcc 5.4.0.

We utilized CGAL [5] for its exact orientation predicates, however, parts of the code are still prone to numerical errors. For example, triangulating the remaining polygonal

Table 1 Diamond test implementation on uniformly distributed point sets. The table shows the mean and the standard deviation of 25 different instances. The extreme values are assumed by points at the point set boundary.

		Number of visited neighbors per point				
n	Edges	Mean	SD	Min	Max	$\mathcal{DS} = 2\pi$
10^1	36.16 ±2.63	9 ±0	0 ±0	9 ±0	9 ±0	0 ±0
10^2	882.8 ±27.69	55.6 ±3.1	16.6 ±2.04	23.72 ±4.82	98.56 ±1.27	30.4 ±4.61
10^3	10,731.7 ±159.9	72.52 ±1.56	23.16 ±1.3	22.68 ±4.55	173 ±14.61	737.84 ±10.91
10^4	$1.1316 \cdot 10^5$ ±471.24	77.64 ±0.69	26.64 ±0.73	19.08 ±2.3	363.72 ±20	9,126.08 ±18.74
10^5	$1.15 \cdot 10^6$ ±1,538.64	72.84 ±0.29	23.76 ±0.47	15.96 ±1.61	846.24 ±24.4	97,200.9 ±40.29
10^6	$1.1562 \cdot 10^7$ ±4,737.67	74 ±0.51	25.76 ±0.39	13.28 ±1.31	2,884.96 ±38.53	$9.9117 \cdot 10^5$ ±61.86
10^7	$1.1579 \cdot 10^8$ ±19,254	77 ±0.6	27.24 ±0.79	11.88 ±0.99	9,567.52 ±78.84	$9.9721 \cdot 10^6$ ±100.61
10^8	$1.1585 \cdot 10^9$ ±56,063.1	72 ±0.94	24.08 ±0.69	10.6 ±0.49	25,017.8 ±107.4	$9.9911 \cdot 10^7$ ±239.64

Table 2 LMT-skeleton statistics on uniformly distributed point sets.

	Possible edges after			Certain edges after		
n	Diamond	LMT	LMT+	LMT	LMT+	Simple Polygons
$1 \cdot 10^1$	36.76 ±2.78	3.8 ±3.84	3.72 ±3.62	19.32 ±2.22	19.32 ±2.22	0.68 ±0.61
$1 \cdot 10^2$	871.92 ±46.37	84.04 ±20.14	74.56 ±18.1	251.48 ±7.12	252.28 ±7.12	10.52 ±2.55
$1 \cdot 10^3$	10,687.4 ±146.68	1,150.32 ±98.05	1,031.96 ±86.46	2,540 ±32.33	2,548.04 ±31.41	128 ±9.2
$1 \cdot 10^4$	$1.1322 \cdot 10^5$ ±661.16	12,637 ±281.25	11,271.76 ±251.6	25,193.44 ±73.29	25,287.56 ±76.43	1,367.08 ±24.65
$1 \cdot 10^5$	$1.1503 \cdot 10^6$ ±1,696.31	$1.2941 \cdot 10^5$ ±1,198.41	$1.1523 \cdot 10^5$ ±973.14	$2.5129 \cdot 10^5$ ±322.29	$2.5227 \cdot 10^5$ ±306.72	13,819.44 ±67.93
$1 \cdot 10^6$	$1.1563 \cdot 10^7$ ±5,459.02	$1.3044 \cdot 10^6$ ±2,708.78	$1.1617 \cdot 10^6$ ±2,486.36	$2.5098 \cdot 10^6$ ±847.61	$2.5194 \cdot 10^6$ ±860.53	$1.3904 \cdot 10^5$ ±232.43
$1 \cdot 10^7$	$1.1579 \cdot 10^8$ ±17,587.01	$1.3074 \cdot 10^7$ ±11,021.75	$1.1645 \cdot 10^7$ ±8,825.57	$2.5088 \cdot 10^7$ ±2,774.11	$2.5184 \cdot 10^7$ ±2,727.23	$1.3931 \cdot 10^6$ ±607.95
$3 \cdot 10^7$	$3.4747 \cdot 10^8$ ±28,678.6	$3.9239 \cdot 10^7$ ±18,919.14	$3.4949 \cdot 10^7$ ±15,068.66	$7.5258 \cdot 10^7$ ±4,637.8	$7.5547 \cdot 10^7$ ±4,563.03	$4.1797 \cdot 10^6$ ±969.6

faces requires to compute and compare the sum of radicals, which we implemented with double-precision arithmetic. For small instances, it was possible to compare the results of our implementation against an independent implementation based on an integer programming formulation of the MWT problem. However, straightforward integer programming becomes infeasible quite fast and comparisons for point sets with thousands of points were not possible.

5.1 Uniformly and normally distributed point sets

Table 1 shows results of our diamond test implementation on uniformly distributed point sets with sizes ranging from 10 to 10^8 points. The table shows the mean values and the standard deviation of 25 different instances. Each instance was generated by choosing n points uniformly from a square centered at the origin. Point coordinates were double-precision values. The diamond test performs one incremental nearest neighbor query for each point in order to generate the edges that pass the test. On average only a small number of neighbors need to be processed for each point. The last column shows the number of queries where all nodes in the spatial tree were discarded because dead sectors covered the whole search space. The numbers show that this is the regular case; the exceptional cases occur at points near the point set "boundary".

Table 2 shows statistics for the LMT-heuristic on uniformly distributed point sets. The instance sizes range from 10 points up to 30,000,000 points. For each size 25 different instances were generated. For the largest instances, the array storing the half-edges consumes nearly 39 GB of memory on its own. The serial initialization of the half-edge data structure, which basically amounts to radially sorting edges, takes longer than the parallel `LMT-Loop` on uniformly and normally distributed points. The improved LMT-skeleton by Aichholzer et al. is denoted LMT+ in the tables. The resulting skeleton was almost always connected in the computations and the number of remaining simple polygons that needed to be triangulated is

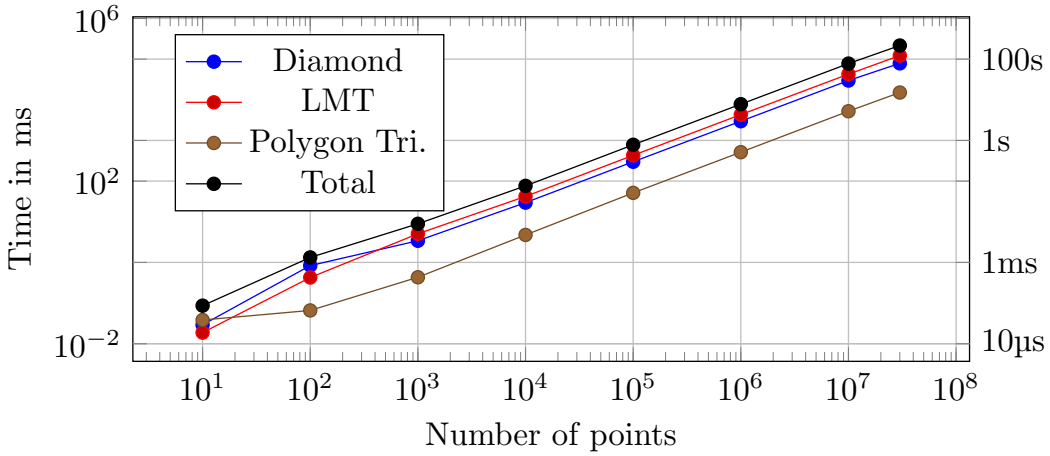


Figure 6 LMT-skeleton runtime on uniformly distributed point sets.

shown in the last column. Only one instance of size $3 \cdot 10^7$ contained one non-simple polygon in the experiments. While developing and optimizing the implementation, we computed the LMT-skeleton of several hundred instances with a million points, without ever seeing a disconnected component.

As we can see, the LMT-skeleton eliminates most of the possible edges with only $\approx 11\%$ remaining. Given that any triangulation has $3n - |\texttt{CH}| - 3$ edges, the certain edges amount to $\approx 83\%$ of the complete triangulation. The improved LMT-skeleton reduces the amount of possible edges by another 10%, but it provides hardly any additional certain edges.

The results on normally distributed point sets are basically identical. Point coordinates were generated by two normally distributed random variables $X, Y \sim \mathcal{N}(\mu, \sigma^2)$, with mean $\mu = 0$ and standard deviation $\sigma \in \{1, 100, 100000\}$. The tables are given in the full version.

5.2 TSPLIB + VLSI

In addition to uniformly and normally distributed instances, we ran our implementation on instances found in the well-known TSPLIB [20], which contains a wide variety of instances with different distributions. The instances are drawn from industrial applications and from geographic problems. All 94 instances have a connected LMT-skeleton and can be fully triangulated with dynamic programming to obtain the minimum weight triangulation. The total time it took to solve all instances of the TSPLIB was approximately 8.5 seconds. A complete breakdown for each instance is given in the full version.

Additional point sets can be downloaded at `http://www.math.uwaterloo.ca/tsp/vlsi/`. This collection of 102 TSP instances was provided by Andre Rohe, based on VLSI data sets studied at the Forschungsinstitut für Diskrete Mathematik, Universität Bonn. The LMT-heuristic is sufficient to solve all instances, except `lra498378`, which contained two non-simple polygonal faces. A complete breakdown is given in the full version. Our implementation of the improved LMT-skeleton performs exceedingly bad on some of these instances; see Table 3. These instances contain empty regions with many points on the "boundary". Such regions are the worst-case for the heuristics because most edges inside them have the diamond property, which in turn leads to vertices with very high degree. Whenever an edge is found to be impossible by the improved LMT-skeleton, almost all edges are restacked and rechecked. Given the overall poor results of the improved LMT-skeleton, storing additional data to increase performance and/or limiting it to non-simple polygons may be reasonable.

Table 3 Statistics for VLSI instances with long runtime.

	Time in ms					
Instance	Total	DT	LMT-Init	LMT-Loop	LMT+	Dyn. Prog.
ara238025	15,325	4,954	446	496	9,279	148
lra498378	382,932	44,267	1,238	7,532	329,292	599
lrb744710	484,430	7,952	1,377	2,661	471,564	872
sra104815	1,937	559	191	198	922	65

6 Conclusion

We have shown that despite of the theoretical hardness of the MWT problem, a wide range of large-scale instances can be solved to optimality.

Difficulties for other instances arise from two sources. On one hand, we have instances containing more or less regular k-gons with one or more points near the center. These configurations can lead to a highly disconnected LMT-skeleton (an example is given by Belleville et al. [4]) and require exponential time algorithms to complete the MWT. Preliminary experiments suggest that such configurations are best solved with integer programming. The example point set given by Belleville et al. [4] can easily be solved with CPLEX in less than a minute, while the dynamic programming implementation of Grantson et al. [14] was not able to solve it within several hours. On the other hand, we have instances containing empty regions with many points on their "boundary", such as empty k-gons and circles. They may be solvable in polynomial time, but trigger the worst-case behavior of the heuristics. Deciding what is the best approach to handle these two types of difficulties and integrating it into our implementation is left for future work.

References

1 Oswin Aichholzer, Franz Aurenhammer, and Reinhard Hainz. New Results on MWT Subgraphs. *Inf. Process. Lett.*, 69(5):215–219, 1999.

2 Ronald Beirouti. A Fast Heuristic for Finding the Minimum Weight Triangulation. Master's thesis, University of British Columbia, Vancouver, BC, Canada, Canada, 1997.

3 Ronald Beirouti and Jack Snoeyink. Implementations of the LMT Heuristic for Minimum Weight Triangulation. In Ravi Janardan, editor, *Symposium on Computational Geometry*, pages 96–105. ACM, 1998.

4 Patrice Belleville, J. Mark Keil, Michael McAllister, and Jack Snoeyink. On Computing Edges That Are In All Minimum-Weight Triangulations. In Sue Whitesides, editor, *Symposium on Computational Geometry*, pages 507–508. ACM, 1996.

5 CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

6 Gautam Das and Deborah Joseph. *Which triangulations approximate the complete graph?*, pages 168–192. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989.

7 Matthew Dickerson, J. Mark Keil, and Mark H. Montague. A Large Subgraph of the Minimum Weight Triangulation. *Discrete & Computational Geometry*, 18(3):289–304, 1997.

8 Matthew Dickerson and Mark H. Montague. A (Usually?) Connected Subgraph of the Minimum Weight Triangulation. In Sue Whitesides, editor, *Symposium on Computational Geometry*, pages 204–213. ACM, 1996.

9 Robert L. Scot Drysdale, Scott A. McElfresh, and Jack Snoeyink. On exclusion regions for optimal triangulations. *Discrete Applied Mathematics*, 109(1-2):49–65, 2001.

10 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

11 P. D. Gilbert. New results in planar triangulations. Master's thesis, University Illinois, 1979.

12 Magdalene Grantson, Christian Borgelt, and Christos Levcopoulos. A Fixed Parameter Algorithm for Minimum Weight Triangulation: Analysis and Experiments. Technical report, Lund University, Sweden, 2005.

13 Magdalene Grantson, Christian Borgelt, and Christos Levcopoulos. Minimum Weight Triangulation by Cutting Out Triangles. In Xiaotie Deng and Ding-Zhu Du, editors, *ISAAC*, volume 3827 of *Lecture Notes in Computer Science*, pages 984–994. Springer, 2005.

14 Magdalene Grantson, Christian Borgelt, and Christos Levcopoulos. Fixed Parameter Algorithms for the Minimum Weight Triangulation Problem. *Int. J. Comput. Geometry Appl.*, 18(3):185–220, 2008.

15 Gisli R. Hjaltason and Hanan Samet. Ranking in Spatial Databases. In *SSD '95: Proceedings of the 4th International Symposium on Advances in Spatial Databases*, pages 83–95, London, UK, 1995. Springer-Verlag.

16 Michael Hoffmann and Yoshio Okamoto. The Minimum Weight Triangulation Problem with Few Inner Points. In Rodney G. Downey, Michael R. Fellows, and Frank K. H. A. Dehne, editors, *IWPEC*, volume 3162 of *Lecture Notes in Computer Science*, pages 200–212. Springer, 2004.

17 G.T. Klincsek. Minimal Triangulations of Polygonal Domains. *Annals of Discrete Mathematics*, 9:121–123, 1980.

18 Bernard M. E. Moret and Henry D. Shapiro. *Algorithms from P to NP (Vol. 1): Design and Efficiency*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

19 Wolfgang Mulzer and Günter Rote. Minimum-weight Triangulation is NP-hard. *J. ACM*, 55(2):11:1–11:29, 2008.

20 G. Reinelt. TSPLIB- a Traveling Salesman Problem Library. *ORSA Journal of Computing*, 3(4):376–384, 1991.